Asignación de Tratamientos a Responsabilidades en el contexto del Diseño Arquitectónico Dirigido por Modelos

David Ameller, Xavier Franch
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3
08034 Barcelona
{dameller, franch}@lsi.upc.edu

Resumen

Una de las principales actividades en el desarrollo de Sistemas de Información (SI) es la asignación de tratamientos a responsabilidades durante la etapa de diseño. En este artículo, presentamos el marco *Responsibility Detection and Transformation* (RDT) para la semi-automatización de este proceso aplicable al contexto del desarrollo dirigido por modelos (MDD). RDT ofrece la posibilidad de partir de diversos tipos de artefactos (especificaciones OO, interfaces gráficas, etc.) para describir el sistema software. Para llevar a cabo el proceso de asignación de tratamientos a responsabilidades, RDT selecciona el tratamiento más adecuado de un repositorio de tratamientos, en base a los requisitos no funcionales establecidos sobre el SI en desarrollo. A continuación, presentamos el prototipo actual de la implementación de RDT, llamado *Assignment of Responsibilities in a 3-Layer architecture* (AR3L), en el que usamos una plataforma de código abierto llamada AndroMDA para implementar el proceso, especificaciones en UML como artefacto a partir del que se extraen las responsabilidades, y arquitectura en tres capas como patrón arquitectónico del SI.

1. Introducción

La fase de diseño de un SI puede contemplarse como una actividad de asignación de tratamientos tecnológicos a las responsabilidades identificadas en el SI. Según [6], "una responsabilidad contiene uno o más propósitos u obligaciones de un elemento", por ejemplo, la responsabilidad de asegurar que no pueden existir en el sistema dos instancias de una clase con un mismo identificador. Por otro lado, un tratamiento es una acción asignada a una responsabilidad con el fin de que ésta sea satisfecha, p.e., definir una clave primaria en el esquema de base de datos nos asegura que la responsabilidad anterior queda satisfecha. Los tipos de responsabilidades son bastante estables y pueden deducirse de diversos tipos de artefactos (especificación OO, interfaz de usuario, etc.). En cambio, el repertorio de tratamientos disponible viene dado por la oferta actual de tecnologías (Hibernate, EJBs, Swing, etc.), lenguajes de programación (Java, C++, C#, etc.), estándares (XML, etc.), …, por lo que tiene un carácter más volátil.

El diseño basado en aplicar tratamientos a responsabilidades fue propuesto a finales de los 80-principios de los 90 [11] y posteriormente adoptado de una manera u otra por diversas metodologías de gran divulgación, especialmente en el ámbito OO, que se basan en esta idea para construir la arquitectura de los SI a partir de su especificación: RUP [5], Larman [6], Fowler [3], etc. En este artículo formulamos una propuesta para semi-automatizar el proceso de asignación de tratamientos a responsabilidades en el marco del desarrollo dirigido por modelos (MDD). El MDD es un paradigma de desarrollo de software que ha ido ganando aceptación en los últimos años. Según [8], "El desarrollo por modelos es simplemente la capacidad de crear un modelo del sistema y convertirlo en algo real". La detección y asignación de responsabilidades se puede considerar un caso del MDD, siendo necesario definir qué es el *modelo del sistema*, qué es *algo real* y cómo se ha implementado la *capacidad*:

- El *modelo del sistema* es un conjunto de artefactos (eventualmente con cierto solapamiento) que describen el comportamiento del sistema. Un ejemplo típico sería una especificación OO que, según

[6], está compuesta de un diagrama de clases con restricciones de integridad, un modelo de comportamiento y un modelo de estado. Pero también la interfaz del usuario, donde la existencia de elementos como desplegables, campos numéricos, etc., determina ciertas responsabilidades a cubrir.

- El *algo real* es el conjunto de tratamientos aplicados a cada responsabilidad detectada en los artefactos de partida. Estos tratamientos pueden ser eventualmente transformados en código pero no trataremos esta problemática en este artículo ya que, como veremos en la próxima sección, nuestro objetivo es obtener un modelo preparado para ser usado por herramientas de MDD capaces de generar código.
- La *capacidad* se ha implementado como un proceso de dos pasos. El primer paso identifica las responsabilidades existentes en los artefactos de partida y el segundo selecciona los tratamientos más adecuados para cumplir los requerimientos no funcionales del sistema. Ambos pasos son semi-automáticos. En el caso de la detección, será más automático mientras más preciso sea el modelo de partida. En el caso de la selección, mientras más detallados sean los requisitos no funcionales.

El resto del artículo se estructura de la manera siguiente. En la sección 2 presentamos el marco genérico RDT. En la sección 3 particularizamos RDT para un caso particular, dando lugar a AR3L. En la sección 4 mostramos un escenario de uso para ilustrar los conceptos principales. En las secciones 5 y 6 presentamos la forma en que trabajamos con responsabilidades y tratamientos. En la sección 7 sumarizamos los detalles de implementación del prototipo actual. Finalmente, en la sección 8 damos un resumen y exponemos las vías de trabajo futuras.

2. El marco RDT

En esta sección presentamos el marco genérico *Responsibility Detection and Transformation* (RDT) para la asignación de tratamientos a responsabilidades. Para ello, vamos a motivar su utilidad en el contexto del MDD y, a continuación, daremos algunos detalles de su arquitectura.

En la figura 1, parte superior, presentamos la arquitectura habitual de los métodos MDD actuales. En resumen, dichas propuestas proponen la generación de código en un lenguaje determinado a partir de un cierto tipo de artefacto de entrada para un patrón arquitectónico concreto [2]. Por ejemplo, AndroMDA o ArcStyler permiten la transformación entre modelos PIM y PSM y la generación de código, limitándose a UML [10] como entrada y Java o .NET como salida. En [7], se propone usar el estándar UsiXML como PIM para diseñar un modelo abstracto y pasando a un modelo concreto, pueden generarse interfaces de usuario orientadas a la Web.

En la figura 1, parte inferior, mostramos el encaje de RDT en dichas propuestas. Destacamos dos características principales. Primero, el marco RDT tiene como objetivo reconocer diversos tipos de artefactos de entrada, tanto modelos como otros artefactos menos formales. En todos los casos, RDT va a absorber todas las responsabilidades que detecte en los artefactos de entrada (que pueden ser uno o varios, redundantes o no, etc.), y va a permitir añadir manualmente aquéllas que no aparezcan en los mismos. Segundo, el marco RDT está diseñado para generar diversos tipos de artefactos de salida (PSM) y así permitir conectarse con diversos generadores de código. Este enfoque posibilita plantearse el uso de las propuestas MDD existentes en más situaciones, por ejemplo usar UsiXML tal como se propone en [7] como artefacto de entrada a partir del que se pueden identificar responsabilidades usadas para generar un PSM UML usable por otras herramientas.

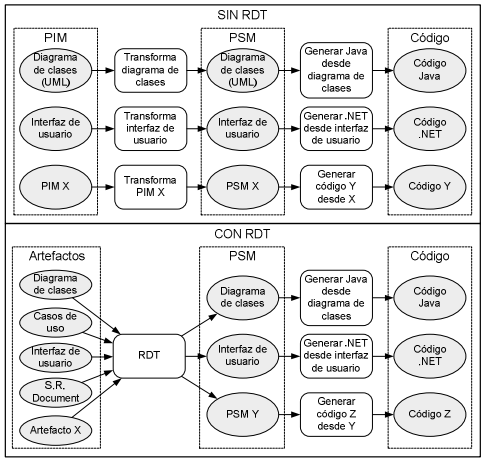

Figura 1: Propuestas MDD sin el marco RDT (parte superior) y con el marco RDT (parte inferior)

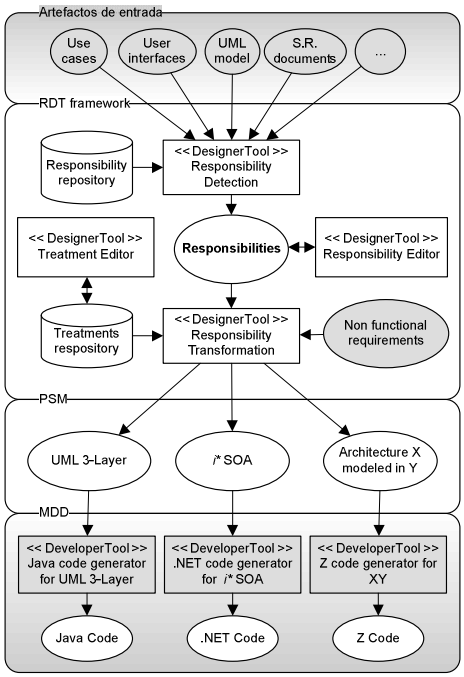

Figura 2: Arquitectura del marco RDT.

Dados los objetivos, queda claro que la arquitectura de RDT debe enfrentarse a dos retos: primero, identificar las responsabilidades inherentes al artefacto o artefactos de partida; segundo, decidir qué tratamientos son más adecuados para estas responsabilidades.

En la figura 2 (evolución de la figura 1, parte inferior) presentamos la arquitectura de RDT[1]. El primer proceso, la detección, depende del tipo de los artefactos usados como punto de partida del diseño. El proceso de detección está diseñado para permitir extraer las responsabilidades de un único tipo de artefacto, p.e. un documento de especificación de requisitos o un diseño de interfaz de usuario, o de varios, por ejemplo una especificación UML que conste de diagrama de clases, modelo de casos de uso y contratos de operaciones [6]. La herramienta *Responsibility Detection* es la encargada de identificar las responsabilidades de dichos artefactos y clasificarlas según los tipos de responsabilidades almacenados en el *Responsibility Repository*. Puede pasar (como veremos en este artículo) que los artefactos deban cumplir ciertas convenciones (p.e., notacionales) para explotar al máximo el proceso de detección automática. En todo caso, se puede prever cierto grado de interacción del diseñador para proporcionar información adicional que permita detectar y/o clasificar las responsabilidades (herramienta *Responsibility Editor*). El resultado del proceso de detección es pues un conjunto de responsabilidades del SI clasificadas por su tipo.

El segundo proceso, la transformación, parte de dicho conjunto de responsabilidades detectadas y las transforma con el fin de que puedan ser gestionadas automáticamente. Esta transformación será efectuada mediante una selección automática de tratamientos a partir de los requisitos no funcionales del sistema software (herramienta *Responsibility Transformation*). Los tratamientos a aplicar residen en un *Treatments Repository* y su aplicación dependerá asimismo del patrón arquitectónico que rige el SI (p.e., tres capas, orientado a servicios, etc.). El modelo resultante puede ser de distintos tipos (UML, *i**, etc), aunque el habitual sería UML, ya que existen más herramientas capaces de transformar modelos en dicho formato. En el supuesto de MDD, las herramientas de generación de código tendrían suficiente información para que el trabajo del desarrollador se redujera considerablemente.

Por último, citemos que la arquitectura de RDT favorece la extensibilidad según diversos criterios: reconocimiento de nuevos tipos de artefactos de entrada o salida, añadido de nuevos tipos de responsabilidades y (especialmente) tratamientos, consideración de nuevos patrones arquitectónicos, etc. Desde un punto de vista conceptual, esta extensibilidad se basa en el uso del concepto de responsabilidad como elemento central en el proceso de diseño.

3. El proyecto AR3L

En esta sección presentamos el proyecto AR3L, que se puede considerar una prueba de concepto del marco RDT para un contexto determinado. Concretamente, AR3L fija: el tipo de artefacto de entrada (especificación UML), el tipo de artefacto de salida (una descripción textual de los tratamientos a aplicar), los tipos de requisitos no

[1] Los recuadros blancos son herramientas que forman parte del marco RDT mientras que las sombreadas son aquéllas que ya existen o que simplemente están fuera de los objetivos principales del proyecto. Las elipses sombreadas son artefactos de entrada mientras que las blancas son información transformada.

funcionales considerados (cuatro, relacionados con la usabilidad y la tecnología) y sus posible valores, y el patrón arquitectónico del SI (arquitectura en tres capas [1] en el marco del diseño OO [6]). La elección de estas variantes se debe a su gran predominancia en el estado del arte actual.

El proyecto AR3L se basa en una herramienta del mismo nombre. Nuestro objetivo ha sido adaptar una plataforma MDD de las muchas existentes [2], AndroMDA, un generador de código basado en una arquitectura dirigida por modelos (MDA) [9]. AndroMDA es una plataforma muy versátil y adaptable; la versión actual, AndroMDA 3.2, da soporte a UML 2.0, está integrada con Eclipse y contiene un conjunto de extensiones que permiten generar código para distintas tecnologías.

En la figura 3, presentamos el prototipo actual de AR3L. La figura refleja que las herramientas de detección y de transformación de responsabilidades se han integrado en una única herramienta, AR3L. En concreto, la detección de responsabilidades se ha implementado usando las facilidades de AndroMDA (en la sección 7 presentamos algún detalle adicional) mientras que la transformación de responsabilidades se ha desarrollado mediante código Java invocado desde AndroMDA. La arquitectura interna de AR3L se ha diseñado lo más desacoplada posible, para que el módulo Java de transformación de responsabilidades pueda reutilizarse en otros contextos.

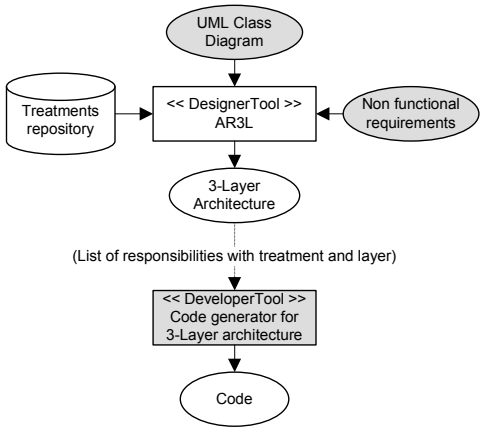

Figura 3: La herramienta AR3L

En esta primera fase de desarrollo, la especificación UML de partida consiste en un diagrama de clases con ciertas restricciones, e.g. asociaciones binarias solamente. Para el modelo de comportamiento, consideramos la existencia de restricciones del tipo pre y poscondiciones que aparecen en las operaciones UML del diagrama de clases. En este prototipo, la salida es simplemente un listado textual de las responsabilidades detectadas con los tratamientos elegidos, por lo que actualmente no puede usarse ningún generador externo de código sin una interpretación previa del resultado generado.

4. Escenario: Gestión de conferencias

En el resto del artículo usaremos un escenario de uso como ejemplo, un sistema gestor de conferencias. Las conferencias están organizadas en sesiones, los artículos se envían a las conferencias en una fecha. Cada artículo enviado se evalúa como aceptado o rechazado, los aceptados se asignan a sesiones de la conferencia a la cual fueron enviados. La figura 4 muestra el diagrama de clases que representa esta información, como es usual algunas restricciones aparecen de forma gráfica (p.e., cardinalidades), mientras que otras como los identificadores de clase deben expresarse textualmente, en este caso los identificadores son: Paper (title), Congress (name o acronym) y Date (date). Además, no puede haber dos sesiones con el mismo nombre en el mismo congreso. Estas restricciones comentadas y otras pertenecientes a las operaciones forman parte de la especificación y del modelo de comportamiento, ya sea con lenguaje natural, OCL u otro mecanismo de especificación.

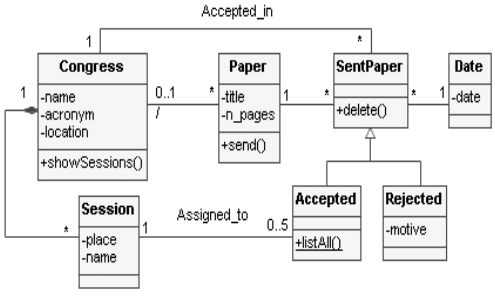

Figura 4: Diagrama de clases del gestor de conferencias

5. Responsabilidades

En el proyecto AR3L, clasificamos las responsabilidades en dos categorías que se derivan de los tipos habituales de restricciones en diagrames de clase UML [4]:

- *Responsabilidades gráficas*. Detectadas en elementos gráficos, como la cardinalidad (de asociaciones, atributos, etc.) y otro tipo de propiedades (herencia, constantes, etc.).
- *Responsabilidades textuales*. No aparecen en el modelo pero están expresadas textualmente con lenguaje natural o formal (p.e., OCL). De estas últimas destacamos dos tipos:
  - *Permanentes*. Deben cumplirse en cualquier estado válido del sistema (p.e., identificadores de clase, restricciones en atributos de clase). Son representados por invariantes de clase.
  - *Eventuales*. Deben cumplirse cuando se invoca una operación. Existen dos tipos, precondiciones y poscondiciones.

En esta prueba de concepto, hemos implementado en AR3L un ejemplo representativo de cada tipo de responsabilidad: cardinalidad de asociaciones (gràfica), identificadores de clase (textual permanente) y precondiciones y poscondiciones de las operaciones (textual eventual). La tabla 1 ofrece las responsabilidades de estos tres tipos que pueden deducirse del modelo UML de la fig. 4.

Tabla 1: Lista de responsabilidades del sistema gestor de conferencias

Responsibility	Model Element Name	Responsibility description
Cardinality	Paper	An object 'Paper' may have at most 1 object 'Congress' bound.
	SentPaper	An object 'SentPaper' must have exactly 1 object 'Paper' bound. An object 'SentPaper' must have exactly 1 object 'Congress' bound. An object 'SentPaper' must have exactly 1 object 'Date' bound.
	Session	An object 'Session' must have exactly 1 object 'Congress' bound. An object 'Session' may have at most 5 objects 'Accepted' bound.
	Accepted	An object 'Accepted' must have exactly 1 object 'Session' bound.
	Evaluation	An object 'Evaluation' must have exactly 1 object 'Person' bound. An object 'Evaluation' must have exactly 1 object 'InEvaluation' bound.
Identifier	Paper	Title is identifier.
	Congress	Name and acronym are both identifiers.
	Date	Date is identifier.
	Session	There cannot be two sessions with the same name in the same congress.
Pre / post	Paper.send	insertElement: post: Create a SentPaper.
	Congress. showSessions	listAll: post: Returns the set of Sessions bound to this Congress. existsElement: pre: The Congress must have some Session bound.
	SentPaper. delete	deleteElement: post: Delete a SentPaper.
	Accepted. listAll	listAll: post: Returns the set of all the Accepted Papers. notEmptyPopulation: pre: There must be some Accepted Person.

Las restricciones conllevan un problema fundamental ya que existen múltiples estrategias de detección (notas, estereotipos, valores etiquetados, etc.), especialmente en el caso de las restricciones textuales. De hecho se pueden usar estrategias distintas dependiendo de la responsabilidad, en este artículo nos hemos basado en las siguientes características para decidir qué estrategia es la más adecuada en cada caso:

- *Esfuerzo del especificador*. Trabajo extra llevado a cabo por el especificador, el cual puede ser medido en forma de cantidad y complejidad de la información añadida.
- *Comprensibilidad*. Cada estrategia requiere una información extra que requiere ser comprendida.
- *Expresividad*. Con esta característica tenemos en cuenta si la estrategia provoca pérdida de información y si se contemplan todas las posibilidades.
- *Estandarización*. Con el fin de que nuestra solución sea aplicable a más ámbitos toman preferencia las estrategias basadas en conceptos estandarizados.
- *Dificultad de implementación*. Estimación del trabajo que se tendrá que hacer al implementar la estrategia en AndroMDA.

5.1. Identificador de clase

La regla de integridad de identificador de clase permite establecer que no puede haber dos instancias de una clase con el mismo valor para un conjunto de atributos (identificador).

Existen tres tipos de identificador:

- *Key*. El identificador esta compuesto por uno o más atributos de la clase.
- *Weak key*. El identificador esta compuesto por uno o más atributos de la clase conjuntamente con el identificador de otra clase, esta segunda clase estaría unida con la primera mediante composición.
- *Alternative key*. Además de tener una *key*, la clase puede tener otro conjunto de atributos que identifican dos instancias distintas.

En el ejemplo (Figura 4) la clase Congress tiene un *Key* y un *Alternative key* (queda al albur del especificador indicar cual de los dos es el alternativo). Por otra parte la clase Session tiene una *Weak key*, su identificador está compuesto por el *Weak key* y el identificador de la clase Congress conjuntamente.

Para mantener una coherencia los tres tipos de identificadores deben cumplir una serie de restricciones, por ejemplo, una clase puede tener un *Key* o un *Weak key*, pero no ambos al mismo tiempo. Estas restricciones deben cumplirse en el modelo de especificación para que éste pueda ser evaluado correctamente.

Existen diversas alternativas para declarar identificadores en un modelo UML. A continuación las presentamos y analizamos. Se basan en tres posibilidades: usar mecanismos de extensión UML (estereotipos o valores etiquetados); incluir información textual en notas o en la clase propiamente (puede ser en texto o en OCL); usar alguna nomenclatura para detectar el tipo de identificador. La figura 5 muestra las diferentes estrategias y la tabla 2 resume nuestro análisis de estas siete estrategias.

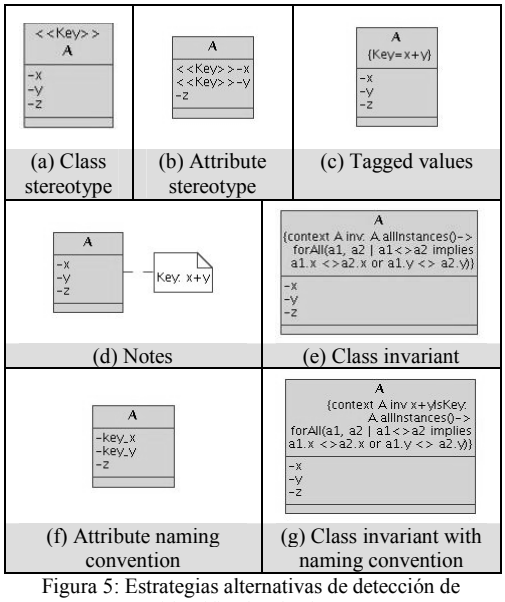

Figura 5: Estrategias alternativas de detección de identificador de clase

- *Class stereotype*. Esta estrategia es muy cómoda para el especificador pero bastante mala en cuanto a expresividad porque no tenemos información de qué atributos componen el identificador. Esto también afecta a la comprensibilidad del modelo.
- *Attribute stereotype*. La clase contiene más información que antes aunque ésta deberá ser introducida por el especificador.
- *Tagged values*. Requiere especificar una sintaxis y obliga al especificador a escribir el nombre de los atributos lo cual puede inducir errores. Además esta estrategia requiere algo más de trabajo en la implementación ya que AndroMDA nos ofrece más facilidades usando estereotipos.
- *Notes*. La primera consecuencia que detectamos es el incremento del tamaño del modelo, por otro lado las notas no son un elemento muy manejable en AndroMDA.
- *Class invariants*. Como ya se ha comentado antes esto supone una serie de problemas en cuanto a la implementación.
- *Attribute naming convention*. Esta estrategia no ofrece una buena comprensibilidad y añadiría otros problemas cuando un atributo forma parte de más de un identificador o cuando hay más de una *Alternative key*.
- *Class invariant with naming convention*. Procesar las expresiones OCL no es tan costoso como antes aunque aún requiere más trabajo que el resto de estrategias.

Tabla 2: Resultado del análisis de estrategias para identificadores de clase

	Facilidad	Expresividad	Comprensibilidad	Estandarización	Implementación
(a)	Alta	Media-Baja	Media-Alta	Media	Alta
(b)	Media-Alta	Alta	Media-Alta	Media	Alta
(c)	Media	Alta	Media-Alta	Media-Baja	Media
(d)	Media	Alta	Media-Baja	Media-Baja	Baja
(e)	Baja	Alta	Baja	Alta	Baja
(f)	Media	Baja	Baja	Media-Alta	Media
(g)	Baja	Alta	Baja	Alta	Media-Baja

Tras este análisis finalmente hemos escogido la estrategia *Atribute Stereotype* porque nos ofrece un buen balance en los cinco criterios evaluados.

5.2. Precondiciones y poscondiciones

Las operaciones que aparecen en el diagrama de clases están descritas por medio de un contrato con restricciones de tipo precondición y poscondición (en este articulo asumimos que los resultados se expresan mediante una poscondición del estilo "post: result = …").

Dada la diversidad de tipos de precondiciones y poscondiciones, para los objetivos del proyecto AR3L nos centramos en un subconjunto suficientemente representativo para la experimentación. Este grupo de restricciones nos permiten analizar cuál es la mejor alternativa en este caso. Las condiciones soportadas (ver ejemplos en la tabla 1) son:

- *Poscondiciones*: Inserción y eliminación de elementos y obtener toda la población de una clase.
- *Precondiciones*: Comprobar la existencia de un elemento y comprobar que existe población.

Igual que en el caso del identificador, el problema principal es escoger qué estrategia usar para detectar las responsabilidades. Básicamente tenemos las mismas opciones que en el caso del identificador teniendo en cuenta que el elemento responsable en este caso es la operación. Otra diferencia importante es que en el caso del identificador teníamos sólo tres subtipos, pero en este caso tenemos muchos más, lo cual dificulta el uso de estereotipos o valores etiquetados. Por ello hemos decidido usar una nomenclatura. Los identificadores usados son: *insertElement*, *deleteElement*, *listAll*, *existsElement* y *notEmptyPopulation*.

5.3. Cardinalidad de asociaciones

En este caso el resto de cardinalidades que aparecen gráficamente podrían tratarse de la misma forma. Antes de continuar cabe remarcar que en nuestro caso sólo tenemos en cuenta las consecuencias de la cardinalidad propiamente sin considerar navegabilidades ya que consideramos que éstas no pertenecen al modelo de especificación.

Cada extremo de una asociación puede inducir una responsabilidad. Dependiendo de la cardinalidad podemos deducir que las cardinalidades "*" y "0" no inducen responsabilidad mientras que las numéricas añaden un limite superior o inferior (o ambos). (p.e., una cardinalidad de tipo "*m..n*" Induce una responsabilidad de mínimo *m* y máximo *n* instancias asociadas)

La cardinalidad de las asociaciones es un ejemplo típico de responsabilidad gráfica que no necesita elementos adicionales ni nomenclaturas, ya que el estándar UML nos ofrece suficiente información. Este tipo de responsabilidades se pueden tratar fácilmente desde AndroMDA y no requieren ningún esfuerzo extra por parte del especificador.

6. Tratamientos

En esta sección mostraremos el comportamiento de AR3L en distintos contextos de uso. Para ello necesitaremos definir algunos tratamientos y criterios no funcionales.

Los tratamientos pueden ser muy diversos. Distinguimos dos tipos, tratamientos estáticos y tratamientos de procesos, los primeros normalmente aparecen en los esquemas de la base de datos, o en tecnologías emergentes como EJB mientras que los tratamientos de procesos son habituales de la capa de dominio aunque también puede aparecer, por ejemplo, en la capa de gestión de datos, por ejemplo, disparadores. En RTD, no prevemos limitación alguna en cuanto al tipo de tratamientos disponibles, pero en AR3L nos hemos focalizado en algunos tratamientos asociados a cada capa para obtener un grupo representativo de una arquitectura en 3 capas (ver tabla 3).

Tabla 3: Lista de tratamientos

Tratamiento	Capa			Descripción
	Presentación	Dominio	Gestión de datos	
InputSizeControl	x			asegura que un conjunto no supera un número determinado de elementos (formulario)
InputFilter	x			asegura que el elemento gestionado existe (combo box)
CardinalityControl		x		asegura que el rol de una asociación tiene un número válido de elementos
Dictionary		x		controla la población de una clase
Hibernate		x	x	tecnología que provee persistencia de forma transparente (patrón Data Mapper [3])
SQL			x	permite manipular directamente la base de datos
Trigger			x	reacciona cuando se viola alguna restricción de integridad
DBSchema			x	declara propiedades en las tablas (primary key)
OnCascade			x	elimina elementos usando borrado en cascada
StoredProcedure			x	permite añadir código en la base de datos

Los tratamientos se pueden aplicar a varias de las responsabilidades detectadas, por ejemplo *InputFilter* para la responsabilidad *ExistsElement*, u *OnCascade* para *DeleteElement*. Cada tratamiento tiene algún efecto sobre los criterios no funcionales, por ejemplo eficiencia, complejidad, etc. Actualmente AR3L asigna los tratamientos dependiendo de los criterios no funcionales, y no es posible aplicar tratamientos distintos a responsabilidades del mismo tipo.

En esta sección tomamos los siguientes criterios no funcionales para la selección de tratamientos:

- *Dependencia tecnológica* (alta, media, baja).
- *Lenguaje de desarrollo* (Java, C++, .NET).
- *Database* (ninguna, Oracle, PostgreSQL).
- *Complejidad de la interficie* (alta, media, baja).

Para establecer la relación entre los criterios no funcionales, los tratamientos y las responsabilidades hemos creado un conjunto de tablas (una por cada tipo de responsabilidad). Uno de los tratamientos de cada tabla será asignado por defecto, lo que nos permite asegurar que todas las responsabilidades tienen como mínimo un tratamiento asignado, que no requieren asumir criterios no funcionales sobre el sistema resultante. La tabla 4 es un ejemplo para el tipo de responsabilidad de cardinalidad: el tratamiento en cursiva es el tratamiento por defecto, las casillas sombreadas son aquéllas que no tienen efecto en el proceso de selección, las "X" impiden el tratamiento y los "√" lo permiten. Los contenidos de la tabla reflejan algunas heurísticas (p.e., un tratamiento que requiere Database padece al menos una dependencia tecnológica media).

Tabla 4: Tabla de tratamientos para la responsabilidad de cardinalidad

Tratamientos para la responsabilidad de cardinalidad de asociaciones	**Criterios no funcionales**											
	Dependencia tecnologica			Lenguaje de desarrollo			Database			Complejidad de la interficie		
	Baja	Media	Alta	C++	Java	.NET	Ninguna	Oracle	Posgres	Baja	Media	Alta
InputSizeControl	X	√	√							X	√	√
CardinalityControl												
Hibernate	X	X	√	X	√	√	X	√	√			
SQL	X	√	√				X	√	√			
Trigger	X	X	√				X	√	√			
DBSchema	X	√	√				X	√	√			

Como ejemplos de uso, consideremos los siguientes (basados en el escenario de uso expuesto en la sección 3):

- Ejemplo 1: Estamos interesados en diseñar un subsistema que obtiene la información de los artículos y automáticamente genera el programa de la conferencia (p.e., la distribución de los artículos en sesiones). No necesitamos base de datos ya que los datos se cargan en memoria. La complejidad de la interficie es mínima ya que no pretendemos interacción con el usuario. Queremos usar el lenguaje C++ y la dependencia tecnología no es relevante.
- Ejemplo 2: Estamos interesados en añadir una interfaz gráfica al subsistema que permita introducir artículos manualmente. Ya que esta interfaz funcionara bajo el sistema operativo Windows, hemos decidido cambiar el lenguaje a .NET.
- Ejemplo 3: Queremos hacer el subsistema persistente, añadiendo una base de datos que mantenga la información de las sesiones y que permita futuras modificaciones.

La tabla 5 muestra un resumen de los criterios no funcionales seleccionados en cada caso.

Tabla 5: Resumen de requisitos no funcionales en los ejemplos vistos

Ejemplo 1	Media	C++	Ninguna	Baja
Ejemplo 2	Media	.NET	Ninguna	Alta
Ejemplo 3	Alta	.NET	Oracle	Alta

La tabla 6 muestra el resultado obtenido de ejecutar AR3L en los distintos contextos descritos. Como se puede ver, en el primer ejemplo AR3L selecciona sólo los tratamientos por defecto para todas las responsabilidades. En el ejemplo 2, se añaden todos los tratamientos de interficie y finalmente en el ejemplo 3, se añaden los tratamientos que requieren una base de datos.

Nótese que con otro conjunto de tratamientos podría darse el caso en el que con los mismos ejemplos obtuviéramos un resultado con más tratamientos disponibles para el primer ejemplo y menos para el último, al contrario del resultado expuesto en la tabla 6.

Tabla 6: Resultados de AR3L en los ejemplos expuestos

			Responsabilidades						
			Identificador de clase	**Cardinalidad de asociaciones**	**Precondiciones y poscondiciones de operaciones**				
					existsElement	*insertElement*	*deleteElement*	*notEmpty Population*	*listAll*
Tratamientos	**Ejemplo 1**	Presentación							
		Dominio	Dictionary	Cardinality Control	Dictionary	Dictionary	Dictionary	Dictionary	Dictionary
		Gestión de datos							
	Ejemplo 2	Presentación		InputSize Control	InputFilter				
		Dominio	Dictionary	Cardinality Control	Dictionary	Dictionary	Dictionary	Dictionary	Dictionary
		Gestión de datos							
	Ejemplo 3	Presentación		InputSize Control	InputFilter				
		Dominio	Dictionary, Hibernate	Cardinality Control, Hibernate	Dictionary, Hibernate	Dictionary, Hibernate	Dictionary, Hibernate	Dictionary, Hibernate	Dictionary, Hibernate
		Gestión de datos	SQL, DBSchema, Hibernate	SQL, Trigger, DBSchema, Hibernate	SQL, Hibernate	SQL, Stored Procedure, Hibernate	SQL, OnCascade, StoredProcedure, Hibernate	SQL, Hibernate	SQL, Hibernate

7. Implementación

Como ya hemos comentado, la implementación de AR3L está basada en una plataforma MDD, AndroMDA. Ésta usa unos componentes llamados *cartuchos* para extender sus funcionalidades. En nuestro *cartucho,* siguiendo las directrices de desarrollo de esta plataforma, hemos extendido el metamodelo de AndroMDA para facilitar la detección de responsabilidades (ver figura 6) y también hemos definido los estereotipos para la detección de identificadores (ver figura 7).

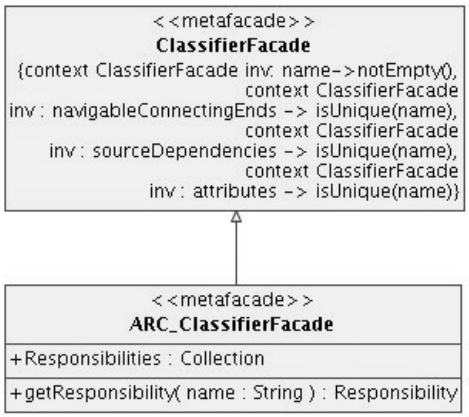

Figura 6: Ejemplo de extensión del metamodelo de AndroMDA

La decisión de extender el metamodelo para definir los estereotipos, como se ve en la figura 7, fue por motivos prácticos. De esta forma AndroMDA nos permite incrustar las restricciones sobre los modelos de entrada en forma de restricciones OCL dentro del propio metamodelo.

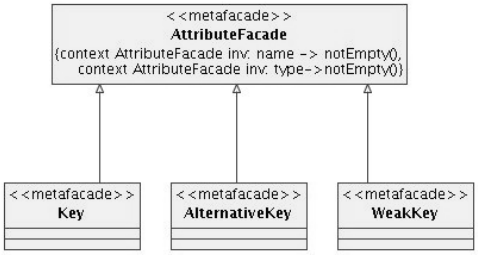

Figura 7: Extensión del metamodelo de AndroMDA para gestionar los estereotipos

Para mejorar la mantenibilidad y extensibilidad del código hemos seguido un criterio de desarrollo: las responsabilidades se detectan desde el elemento responsable, que a su vez es quien almacena sus responsabilidades. Como se puede ver en la figura 6, todas las responsabilidades que pertenecen a una clase quedan almacenadas dentro de su extensión del metamodelo.

En algunos casos no parece la mejor opción determinar las responsabilidades desde el elemento responsable, como en el caso de la responsabilidad gráfica de cardinalidad de asociaciones ya que seria más fácil obtener todas las asociaciones del modelo y ver cuál genera una responsabilidad, en cambio lo que hacemos es tomar todas las clases del modelo (ya que este es el elemento responsable), obtener las terminaciones opuestas de cada una de sus asociaciones y ver si la terminación genera una responsabilidad para dicha clase.

El código fuente de la herramienta AR3L y algunos ejemplos de uso están disponibles en: *http://www.lsi.upc.edu/~gessi/AR3L*

8. Conclusiones y trabajo futuro

En este artículo hemos presentado las ideas genéricas de RDT y el estado actual del proyecto AR3L, un sistema para detectar responsabilidades a partir de la descripción del comportamiento de un sistema y seleccionar tratamientos para éstas acordes con los requisitos no funcionales del sistema. Como resultado, se dispone de una descripción de las tres capas que forman la arquitectura del sistema, a partir de la cual puede aplicarse algún generador de código.

Consideramos que las contribuciones más importantes de RDT son:

- Proponemos un marco de trabajo altamente genérico que permite definir diversos tipos de artefactos de entrada, formatos de generación de salida, y catálogos de responsabilidades y tratamientos. Como resultado, y creemos que es la mayor contribución, el marco de trabajo puede adaptarse a una gran variedad de procesos de desarrollo de software (más rígidos, más ágiles, etc.) y a diversas opciones tecnológicas subyacentes.
- Desacoplamos el diseño arquitectónico propiamente dicho con la generación de código. De esta manera, el diseñador puede experimentar diversas alternativas arquitectónicas y una vez tomada una

decisión, generar código con su herramienta MDD favorita.

- No nos limitamos a una tecnología en concreto (Hibernate, EJBs, etc.), como contrapartida la generación de código es más compleja en esta situación.
- En la implementación actual, AR3L, usamos una plataforma consolidada (AndroMDA) lo cual nos permite sacar partido de los progresos en dicha plataforma.

Esta propuesta puede ser analizada desde distintos puntos de vista:

- Desde el punto de vista del especificador, éste debe incluir más información para obtener buenos resultados con el uso de esta herramienta. Esta cantidad de trabajo extra es relativamente poca y puede ayudar a que la especificación sea más completa.
- Desde el punto de vista del diseñador, éste puede gestionar su catálogo de tratamientos y de criterios no funcionales para estudiar los efectos de sus decisiones, podría interactuar en cada paso refinando los resultados. El diseñador reduciría su volumen de trabajo y además al ser un proceso altamente automatizado se podría cambiar la especificación sin que esto repercutiera muy negativamente en el tiempo de desarrollo del sistema software.
- Desde el punto de vista de un desarrollador, éste tiene a su disposición una arquitectura altamente extensible (p.e., posibilidad de añadir nuevas responsabilidades al proceso de detección), y en la versión actual, mantenible (gracias a la arquitectura de AndroMDA es fácil cambiar un componente por otro).

La dedicación futura a este proyecto se centrará en completar la implementación del RDT Framework (figura 2) y a analizar y solucionar los problemas que no se hayan detectado hasta el momento. Esto incluye la creación de las herramientas complementarias, el soporte de más artefactos de entrada (por ejemplo, en los modelos UML, incluir diagramas de secuencia que declaren las operaciones del sistema), y la generación de diversos tipos de modelos (inicialmente nos centraremos modelos UML por ser estos los más usados) para posibilitar la generación de código automáticamente. Igualmente la definición de requisitos no funcionales y su evaluación jugará un papel importante en el futuro del sistema.

Agradecimientos

Este trabajo se ha desarrollado en el marco del proyecto TIN2004-07461-C02-01.

Referencias

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad: *Pattern Oriented Software Arquitecture*, Vol. 1, John Wiley, 1996.

[2] J. Cabot, E. Teniente: "Constraint Support in MDA Tools: A Survey", In *Procs. 2nd ECMDA*, LNCS 4066, Springer-Verlag, 2006.

[3] M. Fowler: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[4] X. Franch, J. Pradel, J. Raya. "Asignación Sistemática de Responsabilidades en una Arquitectura de Tres Capas". En *Actas XI JISBD*, pp. 293-302, 2006.

[5] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1999.

[6] C. Larman: *Applying UML and Patterns* (3rd edition): Prentice Hall, 2004.

[7] F.J. Martínez-Ruiz, J. Muñoz Arteaga, J. Vanderdonckt, J.M. González-Calleros, R. Mendoza. "A First Draft of a Model-driven Method for Designing Graphical User Interfaces of Rich Internet Application", En *Procs. 4th LA-WEB*, 2006.

[8] S.J. Mellor, A.N. Clark, T. Futagami: "Model-Driven Development", *IEEE Software*, vol. 20, no. 5, pp. 14-18, Sept. 2003.

[9] OMG: "Model Driven Architecture (MDA)". Object Management Group, ormsc/2001-07-01, July 2001.

[10] J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2004.

[11] R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*, Prentice Hall, 1990.